Journal of Cryptographic Engineering
https://doi.org/10.1007/s13389-023-00315-3

REGULAR PAPER

A side-channel attack on a masked and shuffled software implementation of Saber

Kalle Ngo[1] · Elena Dubrova[1] · Thomas Johansson[2]

Received: 12 June 2022 / Accepted: 28 February 2023

Abstract
In this paper, we show that a software implementation of IND-CCA-secure Saber key encapsulation mechanism protected by first-order masking and shuffling can be broken by deep learning-based power analysis. Using an ensemble of deep neural networks trained at the profiling stage, we can recover the session key and the secret key from $257 \times N$ and $24 \times 257 \times N$ traces, respectively, where N is the number of repetitions of the same measurement. The value of N depends on the implementation of the algorithm, the type of device under attack, environmental factors, acquisition noise, etc.; in our experiments $N = 10$ is sufficient for a successful attack. The neural networks are trained on a combination of 80% of traces from the profiling device with a known shuffling order and 20% of traces from the device under attack captured for all-0 and all-1 messages. "Spicing" the training set with traces from the device under attack helps us minimize the negative effect of inter-device variability.

Keywords Public-key cryptography · Post-quantum cryptography · Saber KEM · LWE/LWR-based KEM · Side-channel attack · Power analysis

1 Introduction

Public-key cryptographic schemes used today depend on the intractability of certain mathematical problems such as integer factorization or the discrete logarithm. However, if large-scale quantum computers become a reality, it will be possible to solve these problems in polynomial time using Shor's algorithm [51]. Even though it will take many years to construct a large-scale quantum computer, the need for long-term security makes it urgent to investigate new solutions.

To address this need, the National Institute of Standards and Technology (NIST) started in 2016 a process for standardization of post-quantum cryptographic primitives, NIST PQC. Candidate primitives rely on problems that are not known to be targets for a quantum computer, such as lattices problems and decoding problems in Hamming metric.

In rounds 1 and 2 of the NIST PQC process, security and implementation aspects were the main priority in assessment of the candidates. In round 3, the focus has shifted to side-channel attack resistance. Lattice-based schemes have received a particular attention because, among the finalists of the round 3 for the primitive key encapsulation mechanism (KEM), three out of four were lattice-based: NTRU-based scheme NTRU [15]; Learning With Errors (LWE)-based scheme Kyber [4]; and the Learning With Rounding (LWR)-based scheme Saber [18]. The hardness in these problems comes from inserting unknown noise into otherwise linear equations.

At present the NIST PQC process has entered its fourth round. The resistance to side-channel attacks remains an important criterion for evaluating the candidates. Side-channel attack is considered as a main security threat to implementations of cryptographic algorithms, in particular for applications in embedded devices.

Side-channel attacks were introduced by Kocher et al. [30]. They exploit information obtained from physically measurable, non-primary channels. The most basic form is the timing channel which is exploited by measuring the execution time of software implementations of the cryp-

✉ Kalle Ngo
kngo@kth.se

Elena Dubrova
dubrova@kth.se

Thomas Johansson
thomas.johansson@eit.lth.se

[1] KTH Royal Institute of Technology, Stockholm, Sweden

[2] Lund University, Lund, Sweden

Published online: 25 April 2023

tographic algorithms [31]. The general protection method against timing attacks is to make implementations such that all instructions are executed in constant time, a standard assumption for software implementations. The timing channel can be extended to consider cache-timing attacks [11], where time variation due to memory management in the executing device is considered. A typical example of an exploit is the use of look-up tables.

Even with constant time implementations and avoiding implementation weaknesses such as the use of look-up tables, a software implementation is still vulnerable to attacks if the power consumption or electromagnetic (EM) emissions from the CPU can be measured [1, 30]. In such cases, more advanced countermeasures are required. The main tools are techniques such as masking [14], shuffling [56], insertion of random delays through dummy operations [16], constant-weight encoding [33] and code polymorphism [7].

Differential side-channel analysis pioneered by Kocher et al. [30] was the first breakthrough in the area. The second major advance was the introduction of deep learning-based side-channel analysis. Apart from improving the differential attacks' effectiveness (e.g., four instead of 400 power measurements are needed to extract a key from a USIM [10]), the latter enabled non-differential message/key recovery attacks on NIST PQC candidates [40, 52, 54], as well as attacks of true random number generators [39] and Physical Unclonable Functions [63]. Deep learning-based side-channel attacks can overcome traditional countermeasures, including Boolean masking [40], jitter [12] and code polymorphism [34]. *Our contributions* In this paper, an extension of ASHES'21 [41], we present the first side-channel attack on a **masked and shuffled** implementation of CCA-secure Saber KEM. Additionally, in this extension, we delve deeper into the workings of neural network models and pinpoint the specific assembly instructions that leak information, thereby identifying potential areas for future protection.

Until now, these countermeasures combined together were believed to provide an adequate protection against power and EM analysis.

We show how to recover the session key and the long-term secret key by deep learning-based power analysis from $257 \times N$ and $24 \times 257 \times N$ traces, respectively, captured using the execution of the decapsulation algorithm, where N is the number of repetitions of the same measurement. The value of N depends on the implementation, environmental factors, acquisition noise, etc.; in our experiments, $N = 10$ is enough for a successful attack without any enumeration.

Similarly to the attack on a first-order masked Saber [40], our deep neural networks learn a higher-order model directly, without explicitly extracting random masks at each execution. However, since we attack an implementation in which the message bits are shuffled, it is not possible to directly recover the message from a single trace, as in [40]. Only the message Hamming weight (HW) can be derived. To find the order of message bits, traces for 256 additional decapsulations have to be captured and analyzed for each chosen chiphertext (hence $\times 257$).

We quantify the success rate of the message HW recovery as a function of the success rate of a single message bit recovery and show that the latter should be of the order of 0.999 to recover the message HW with a high probability. To increase the success rate of a single message bit recovery, we introduce a novel approach for training neural networks which uses a combination of traces from the profiling device with a known shuffling order and traces from the device under attack captured for all-0 and all-1 messages. We also use an ensemble of models to increase the success rate of message recovery from the derived HWs.

The remainder of this paper is organized as follows. Section 3 gives the necessary background on Saber algorithm and profiled side-channel attacks. Section 4 describes the implementation of masked and shuffled Saber KEM which is used in our experiments. Section 5 presents equipment for trace acquisition. Section 6 shows how points of interest are located in side-channel measurements. Sections 7 and 8 describe the profiling and the attack stages, respectively. Section 9 summarizes the experimental results. Section 10 concludes the paper and describes future work.

2 Previous work

In this section, we describe previous work on implementations and attacks of the NIST PQC lattice-based candidates.

2.1 Implementations

The first side-channel protected implementation of a lattice-based cryptosystem was presented in [49] followed by [48], based on masking. Masking involves doing linear operations twice, whereas nonlinear operations need more complex solutions decreasing the speed substantially. The implementation approach in [49] increases the number of CPU cycles on an ARM Cortex-M4 by a factor more than 5 compared to a standard implementation, see [6, p. 2].

These protected implementations focus on Chosen-Plaintext Attack (CPA)-secure lattice schemes, but more relevant are secure primitives designed to withstand Chosen-Ciphertext Attacks (CCA). CCA-secure primitives are usually obtained from a CPA secure primitive using a transform, such as the Fujisaki-Okamoto (FO) transform or some variation of it [28]. The CCA-transform is itself susceptible to side-channel attacks and should be protected [47]. Examples of recent masked implementations are: [42] of a KEM similar to NewHope; and [5, 22, 35] being lattice-based signature schemes.

At the time the work presented in this paper was carried out, only one of the round 3 finalists of the NIST PQC, Saber, had a protected software implementation available [6]. The implementation utilizes a first-order masking of the Saber CCA-secure decapsulation algorithm with an overhead factor of only 2.5 compared to an unmasked implementation. This side-channel secure version can be built with relatively simple building blocks compared to other candidates, resulting in a small overhead. The masked implementation of Saber is based on masked logical shifting on arithmetic shares and a masked binomial sampler. The work includes experimental validation of the implementation to confirm it on the Cortex-M4 general-purpose processor.

2.2 Attacks

Early side-channel attacks on NIST PQC project candidates targeted unprotected implementations. In [52] message recovery attacks on the unprotected encapsulation part of round 3 candidates CRYSTALS-Kyber and Saber and round 3 alternate candidate FrodoKEM using a single power trace were described. In [47], near-field EM side-channel assisted chosen ciphertext attacks applicable to six round 2 candidates were presented. In [62], unprotected Kyber was attacked as a case study using near-field EM side-channels. A way of turning a message recovery attack to a secret key recovery attack was proposed using, e.g., 184 traces for 98% success rate. In [55] another power/EM-based secret key recovery attack on some round 3 candidates KEMs based on FO transform and its variant was presented. In [25], similar ideas were used for timing attacks. The resistance of an unprotected Saber to amplitude-modulated EM emanations was investigated in [60] and [59].

In [45], the authors improve the key recovery attacks on unprotected implementations of three NIST PQC finalists, including Saber. They also discuss how to attack masked implementations by attacking shares individually. However, no actual attack on masked Saber was carried out. The first attack on a first-order masked implementation of the IND-CCA-secure Saber KEM was demonstrated in [40]. The attack recovers both the session key and the secret key using a deep neural network trained at the profiling stage. The chosen ciphertext-based secret key recovery attack requires 24 traces. The ciphertexts are constructed using a novel error-correction code-based method which allows for correcting single-bit errors an detecting double-bit errors in the recovered messages. This waves the requirement for a perfect message recovery, making the attack more realistic. An attack applying the method of [40] to a first-order masked implementation of Kyber was presented in [58], targeting the message encoding vulnerability found in [52].

More recently, in [8], side-channel attacks on two implementations of masked polynomial comparison were demonstrated on the example of Kyber. Polynomial multiplication is also a target of the attacks on unprotected implementations of all lattice-based NIST PQC finalists presented [37], where Correlation Power Analysis is used.

3 Background

This section describes Saber and profiled side-channel attacks. A more detailed description of Saber can be found in [18].

3.1 Saber design description

Saber is a package of cryptographic algorithms whose security relies on the hardness of the Module Learning With Rounding problem (Mod-LWR) [18]. It contains a CPA-secure public key encryption scheme, Saber.PKE and a CCA-secure key encapsulation mechanism, Saber.KEM, based on a post-quantum version of the Fujisaki-Okamoto transform [21].

Pseudo-codes of Saber.PKE and Saber.KEM are shown in Figs. 1 and 2, respectively. We follow the notation of [40].

Saber.PKE.KeyGen()

1: $seed_{\mathrm{A}} \leftarrow \mathcal{U}(\{0,1\}^{256})$
2: $\mathbf{A} = \mathrm{gen}(seed_{\mathrm{A}}) \in R_q^{l \times l}$
3: $r \leftarrow \mathcal{U}(\{0,1\}^{256})$
4: $\mathbf{s} \leftarrow \beta_\mu(R_q^{l \times 1}; r)$
5: $\mathbf{b} = ((\mathbf{A}^T\mathbf{s} + \mathbf{h}) \mod q) \gg (\epsilon_q - \epsilon_p) \in R_p^{l \times 1}$
6: **return** $(pk := (seed_{\mathrm{A}}, \mathbf{b}), sk := \mathbf{s})$

Saber.PKE.Dec($\mathbf{s}, (\boldsymbol{c_m}, \mathbf{b}')$)

1: $v = \mathbf{b}'^T(\mathbf{s} \mod p) \in R_p$
2: $m' = ((v + h_2 - 2^{\epsilon_p - \epsilon_T}\boldsymbol{c_m}) \mod p) \gg (\epsilon_p - 1) \in R_2$
3: **return** m'

Saber.PKE.Enc($(seed_{\mathrm{A}}, \mathbf{b}), m; r$)

1: $\mathbf{A} = \mathrm{gen}(seed_{\mathrm{A}}) \in R_q^{l \times l}$
2: **if** r is not specified **then**
3: $\quad r \leftarrow \mathcal{U}(\{0,1\}^{256})$
4: **end if**
5: $\mathbf{s}' \leftarrow \beta_\mu(R_q^{l \times 1}; r)$
6: $\mathbf{b}' = ((\mathbf{A}\mathbf{s}' + \mathbf{h}) \mod q) \gg (\epsilon_q - \epsilon_p) \in R_p^{l \times 1}$
7: $v' = \mathbf{b}^T(\mathbf{s}' \mod p) \in R_p$
8: $\boldsymbol{c_m} = ((v' + h_1 - 2^{\epsilon_p - 1}m) \mod p) \gg (\epsilon_p - \epsilon_T) \in R_T$
9: **return** $(c := (\boldsymbol{c_m}, \mathbf{b}'))$

Fig. 1 Description of Saber.PKE from [18]

Saber.KEM.KeyGen()
1: $(seed_{\mathbf{A}}, \mathbf{b}, \mathbf{s})$ = Saber.PKE.KeyGen()
2: $pk = (seed_{\mathbf{A}}, \mathbf{b})$
3: $pkh = \mathcal{F}(pk)$
4: $z \leftarrow \mathcal{U}(\{0,1\}^{256})$
5: **return** $(pk := (seed_{\mathbf{A}}, \mathbf{b}), sk := (z, pkh, pk, \mathbf{s}))$

Saber.KEM.Encaps($(seed_{\mathbf{A}}, \mathbf{b})$)
1: $m \leftarrow \mathcal{U}(\{0,1\}^{256})$
2: $(\hat{K}, r) = \mathcal{G}(\mathcal{F}(pk), m)$
3: c = Saber.PKE.Enc$(pk, m; r)$
4: $K = \mathcal{H}(\hat{K}, c)$
5: **return** (c, K)

Saber.KEM.Decaps($(z, pkh, pk, \mathbf{s})$,c)
1: m' = Saber.PKE.Dec$(\mathbf{s}, c)$
2: $(\hat{K}', r') = \mathcal{G}(pkh, m')$
3: c' = Saber.PKE.Enc$(pk, m'; r')$
4: **if** $c = c'$ **then**
5: **return** $K = \mathcal{H}(\hat{K}', c)$
6: **else**
7: **return** $K = \mathcal{H}(z, c)$
8: **end if**

Fig. 2 Description of Saber.KEM from [18]

Table 1 Proposed parameters of round 3 Saber

	l	n	q	p	T	μ	Security	p_{fail}
LightSaber	2	256	2^{13}	2^{10}	2^3	10	NIST-I	2^{-120}
Saber	3	256	2^{13}	2^{10}	2^4	8	NIST-III	2^{-136}
FireSaber	4	256	2^{13}	2^{10}	2^6	6	NIST-V	2^{-165}

Let $\mathbb{Z}_q$ be the ring of integers modulo q and R_q be the quotient ring $\mathbb{Z}_q[X]/(X^n + 1)$. The rank of the module is denoted by l. The rounding modulus is denoted by p.

The notation $x \leftarrow \chi(S)$ stands to denote sampling x according to a distribution χ over a set S. The uniform distribution is denoted by $\mathcal{U}$. The centered binomial distribution with parameter μ is denoted by β_μ, where μ is an even positive integer. The term $\beta_\mu(R_q^{l\times k}; r)$ generates a matrix in $R_q^{l\times k}$ where the coefficients of polynomials in R_q are sampled in a deterministic manner from β_μ using seed r.

The functions $\mathcal{F}$, $\mathcal{G}$ and $\mathcal{H}$ are SHA3-256, SHA3-512 and SHA3-256 hash functions, respectively. The gen is an extendable output function which is used to generate a pseudorandom matrix $\mathbf{A} \in R_q^{l\times l}$ from $seed_{\mathbf{A}}$. It is instantiated with SHAKE-128.

The bitwise right shift operation is denoted by "$\gg$". It is extendable to polynomials and matrices by performing the shift coefficient-wise. To allow for an efficient implementation, Saber design uses power-of-two moduli q, p, and T, namely $q = 2^{\epsilon_q}$, $p = 2^{\epsilon_p}$ and $T = 2^{\epsilon_T}$. In order to implement rounding operations by a simple bit shift, three constants are used: polynomials $h_1 \in R_q$ and $h_2 \in R_q$ with all coefficients being $2^{\epsilon_q-\epsilon_p-1}$ and $2^{\epsilon_p-2} - 2^{\epsilon_p-\epsilon_T-1} + 2^{\epsilon_q-\epsilon_p-1}$, respectively, and a constant vector $\mathbf{h} \in R_q^{l\times 1}$ in which each polynomial is equal to h_1.

In the round 3 Saber document [18], three sets of parameters are proposed for the security levels of NIST-I, NIST-III and NIST-V: LightSaber, Saber and FireSaber, respectively (See Table 1). All results presented in this paper are for Saber, but it is trivial to extend them to the other versions. Saber uses $n = 256$, $l = 3$, $q = 2^{13}$, $p = 2^{10}$, $T = 2^4$ and $\mu = 8$. Its decryption failure probability is bounded by 2^{-136}.

3.2 Profiled side-channel attacks

Side-channel attacks can be carried out in two settings: profiled and non-profiled. *Profiled* attacks first learn a leakage profile of the targeted cryptographic algorithm's implementation using a device similar to the device under attack, called *profiling device*. The profiling can be done by creating a template [3, 13, 27], or training a neural network model [10, 12, 29, 32]. Then, the resulting template/model is used to recover the secret variable, e.g., the key, from the device under attack [32]. *Non-profiled* attacks attack directly [53].

Profiled side-channel attacks typically assume that:

(1) The attacker has at least one profiling device similar to the device under attack which runs the same implementation.
(2) The attacker has full control over the profiling device.
(3) The attacker has direct physical access to the device under attack to measure side-channel signals for chosen inputs.

4 Implementation of masked and shuffled Saber KEM

All experiments presented in this paper are performed on a first-order masked and shuffled implementation of Saber which we created ourselves. To the best of our knowledge, no implementations of Saber protected by both masking and shuffling countermeasures are available at present.

We used the first-order masked implementation of Saber presented in [6] as a base and added shuffling on the top as described Sect. 4.2.

4.1 Masking

Masking is a well-known countermeasure against power/EM analysis [14]. *First-order* masking protects against attacks leveraging information in the first-order statistical moment. A first-order masking partitions any sensitive variable x into two shares, x_1 and x_2, such that $x = x_1 \circ x_2$, and executes all

```
void indcpa_kem_dec_masked(uint16 sksv1[],
uint16 sksv2[], char *ct, char m1[], char m2[])
uint16 pksv[K][N];
uint16 v1[N]={0}, v2[N]={0};

1: SABER_un_pack(&ct,v1);
2: for (i = 0; i < N; i++) do
3:    v1[i] = h2-(v1[i]«(EP-ET));
4: end for
5: BS2POLVEC(ct,pksv,P);
6: InnerProd(pksv,sksv1,P-1,v1);
7: InnerProd(pksv,sksv2,P-1,v2);
8: poly_A2A(v1,v2);
9: POL2MSG(v1,m1);
10: POL2MSG(v2,m2);


void poly_A2A(uint16 A[N], uint16 R[N])
uint32 A, R;

1: for (i = 0; i < N; i++) do
2:    A = A[i]; R = R[i];
3:     ... /* processing */
4:    A[i] = A; R[i] = R;
5: end for


void FY_Gen(uint8* fylist, int max)

1: for (i = 0; i < max; i++) do
2:    fylist[i] = i
3: end for
4: for (i = max-1; i > 0; i--) do
5:    int index = rand() % (i+1);
6:    uint8 temp = fylist[index];
7:    fylist[index] = fylist[i];
8:    fylist[i] = temp;
9: end for
```

```
void  indcpa_kem_dec_masked_and_shuffled(uint16  sksv1[],
uint16 sksv2[], char *ct, char m1[], char m2[])
uint16 pksv[K][N];
uint16 v1[N]={0}, v2[N]={0};

1: SABER_un_pack(&ct,v1);
2: for (i = 0; i < N; i++) do
3:    v1[i] = h2-(v1[i]«(EP-ET));
4: end for
5: BS2POLVEC(ct,pksv,P);
6: InnerProd(pksv,sksv1,P-1,v1);
7: InnerProd(pksv,sksv2,P-1,v2);
8: poly_A2A_shuffled(v1,v2);
9: POL2MSG_shuffled(v1,m1);
10: POL2MSG_shuffled(v2,m2);

void poly_A2A_shuffled(uint16 A[N], uint16 R[N])
uint8 fylist[256];
uint32 A, R;

1: FY_Gen(fylist, 256);
2: for (i = 0; i < N; i++) do
3:    y = fylist[i]
4:    A = A[y]; R = R[y];
5:     ... /* processing */
6:    A[y] = A; R[y] = R;
7: end for

void POL2MSG_shuffled(uint16 *v, chair *m)
uint8 fylist[32];
1: FY_Gen(fylist, 32);
2: for (j = 0; j < BYTES; j++) do
3:    y = fylist[j]
4:    m[y] = 0;
5:    for (i = 0; i < 8; i++) do
6:       m[y] = m[y]|(v[8×y+i]«i);
7:    end for
8: end for
```

Fig. 3 The masked implementation of Saber.PKE.Dec() from [6] (left) and the presented masked and shuffled implementation of Saber.PKE.Dec() (right)

operations separately on the shares. The operator "$\circ$" depends on the type of masking, e.g., it is "$+$" in arithmetic masking and "$\oplus$" in Boolean masking.

Carrying out operations on the shares x_1 and x_2 prevents leakage of side-channel information related to x as computations do not explicitly involve x. Instead, x_1 and x_2 are linked to the leakage. Since the shares are randomized at each execution of the algorithm, they are not expected to contain exploitable information about x. The randomization is usually done by assigning a random mask r to one share and computing the other share as $x-r$ for arithmetic masking or $x \oplus r$ for Boolean masking.

A challenge in masking lattice-based cryptosystems is the integration of bitwise operations with arithmetic masking which requires methods for secure conversion between masked representations. Saber can be efficiently masked due to specific features of its design: power-of-two modulo q, p and T, and limited noise sampling of LWR. Due to the former, modular reductions are basically free. The latter implies that only the secret key $\mathbf{s}$ has to be sampled securely. In contrast, LWE-based schemes also need to securely sample two additional error vectors.

Masking duplicates most linear operations, but requires more complex routines for nonlinear operations. The first-order masked implementation of Saber presented [6] uses a custom primitive for masked logical shifting on arithmetic shares, called `poly_A2A()`, and an adapted masked binomial sampler from [50]. Particular attention is devoted in [6] to the protection of the decapsulation algorithm since it involves operations with the long-term secret key $\mathbf{s}$. At its first step (see Saber.KEM.Decaps() in Fig. 2), the decapsulation algorithm calls Saber.PKE.Dec() to decrypt the input ciphertext c. Figure 3 shows the implementa-

tion of Saber.PKE.Dec() from [6] called `indcpa_kem_dec_masked()`.

To perform masked logical shifting, the authors of [6] recognize that, for power-of-two moduli, the conventional method of first performing an A2B conversion and then shifting subsequently the Boolean shares is wasteful. This is because the lower bits are first computed only to be immediately discarded by shifting them out. Their novel primitive, `poly_A2A()` (see Fig. 3), avoids computing the Boolean sharing of the lower bits completely, leading to reduced computational and memory overheads.

4.2 Shuffling

Shuffling is another well-known countermeasure against power/EM analysis [56]. We use the modernized version of the Fisher-Yates (FY) algorithm [20] which generates a random permutation of a finite sequence. The generated sequence is used as the loop iterator to index the inner loop function's data processing. This effectively scrambles the order in which the elements of an array are processed as opposed the linear sequence of a non-shuffled loop. Shuffling makes power analysis and neural network training significantly more difficult as this removes the linear correlation of index sequence with time.

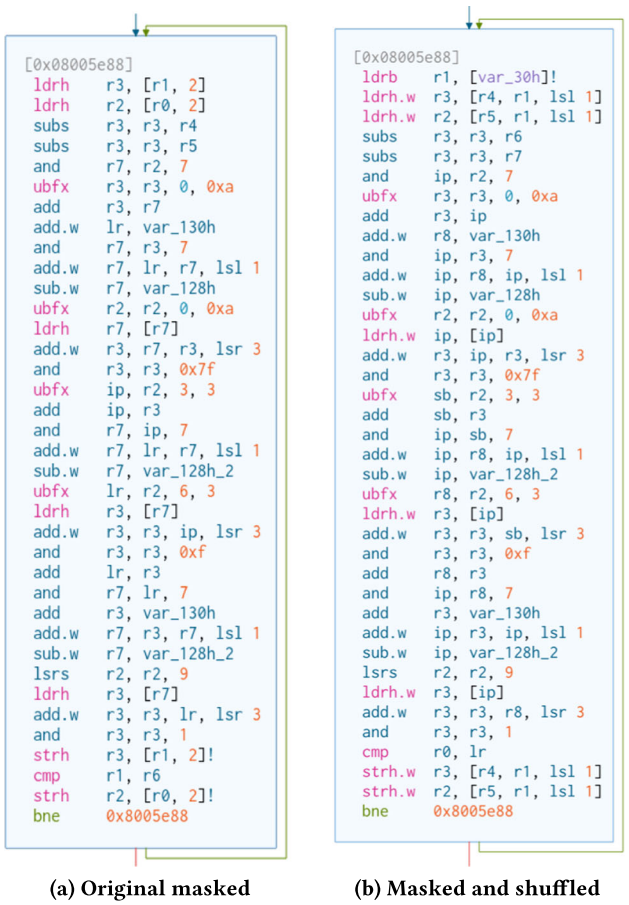
Figure 3 shows our masked and shuffled implementation of the decryption algorithm Saber.PKE.Dec(), called `indcpa_kem_dec_ masked_and_shuffled()`. We implement bitwise shuffling of a 256-bit message in the primitive `poly_A2A()` by calling the `FY_Gen()` function to randomly permute a list of the same length (see `poly_A2A_shuffled()`). The shuffled values (in the range from 0 to 255) are then subsequently referenced at the start of every loop iteration, resulting in randomized execution order.

Figure 4a and b compares the inner loops of the assembly code of `poly_A2A` procedure before and after adding shuffling on the top of masking. One can see that the inclusion of `FY_Gen()` function has a minimal effect. It changes the lines that reference the store/load offsets only. Therefore, the side-channel leakage which is not related to FY index generation is expected to be similar in both implementations.

We also implement bytewise shuffling of a message in the procedure `POL2MSG()` (see `POL2MSG_shuffled()`) by calling the `FY_Gen()` to randomly permute a list of the length equal to the number of bytes, 32.

4.3 Known vulnerabilities

In previous work, a number of vulnerabilities were discovered in the non-masked LWE/LWR-based PKE/KEMs [2, 37, 44–47, 52, 52, 55]. One is Incremental-Storage vulnerability resulting from an incremental update of the decrypted message in memory during message decoding [45]. The decoding



(a) Original masked **(b) Masked and shuffled**

Fig. 4 Assembly code of masked poly_A2A inner loop before and after adding shuffling on the top

function iteratively maps each polynomial coefficient into a corresponding message bit, thus computing the decrypted message one bit at a time.

It was further observed in [45] that a non-masked implementation of the decoding function contains two points with exploitable Incremental-Storage vulnerability. The first one is where the message bits are computed and stored in a 16-bit memory location in an unpacked fashion. Since the memory location can take only two possible values, 0 or 1, an attacker can recover the message bit by distinguishing between 0 and 1. The second point is in `POL2MSG()` procedure where the decoded message bits are packed into a byte array in memory. There has been many attacks put forth against the Fujisaki-Okamoto (FO) transform commonly found in lattice-based KEMs, such as [26, 54] as well as [9] which targets a masked version of the FO's comparison operation through a collision attack.

In [40], it was demonstrated that, despite partitioning the message into two shares in a first-order masked implementation of Saber, the leakage point in `POL2MSG()` procedure can still be exploited. In addition, a new leakage point in `poly_A2A()` procedure was discovered (highlighted in red

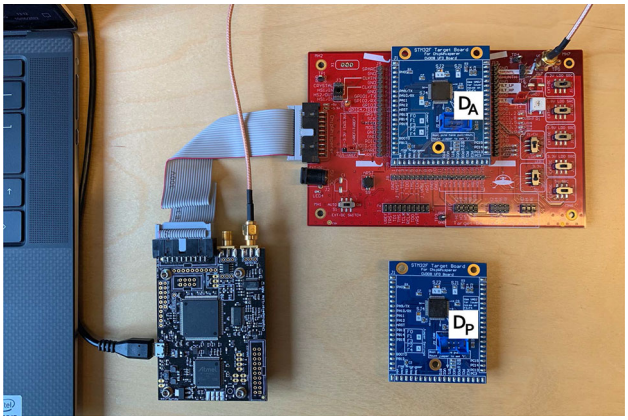


Fig. 5 Equipment for trace acquisition

in Fig. 3). The attacks presented in this paper are based on the corresponding point in `poly_A2A_shuffled()`) (highlighted in red in Fig. 3).

5 Equipment for trace acquisition

The equipment we use for trace acquisition consists of the ChipWhis- perer-Lite board, the CW308 UFO board and two CW308T-STM32F4 target boards (see Fig. 5).

The ChipWhisperer is a hardware security evaluation toolkit based on a low-cost open hardware platform and an open-source software [38]. It can be used to measure power consumption and to make communication between the target device and the computer easier. Power is measured over a shunt resistor connected between the power supply and the target device. ChipWhisperer-Lite employs a synchronous capture method, which greatly improves trace synchronization while also lowering the required sample rate and data storage.

The CW308 UFO board is a general-purpose platform for evaluating multiple targets [17]. The target board is plugged into a dedicated U connector.

The target board CW308T-STM32F4 contains a 32-bit ARM Cortex-M4 CPU with STM32F415-RGT6 device. The board operates at 24 MHz and it is sampled at 24 MHz, i.e., 1 point per clock cycle.

In our experiments, the Cortex-M4 CPU is programmed with the masked and shuffled Saber implementation described in the previous section. The implementation is compiled with `arm-none-eabi- gcc` at the highest level of compiler optimization `-O3` (recommended default) which is typically the most difficult to break by side-channel analysis [52].

6 Locating points of interest

The attacks on unprotected implementations of LWE/LWR-based KEMs [47, 52] typically locate leakage points in side-channel measurements using techniques such as Test Vector Leakage Assessment (TVLA) [24], or Correlation Power Analysis (CPA). However, such a method is not applicable to a protected implementation since masked implementations change random masks for each execution and shuffled implementations change shuffling order for each execution.

In this section, we describe our method for locating points of interest in a masked and shuffled implementation of Saber. Figure 6a shows a power trace obtained by averaging 50K measurement made during the execution of Saber.KEM.Decaps() for random ciphertexts. We can clearly see different blocks with different structure. Our aim is `poly_A2A()` procedure which processes 256 message bits one-by-one. The segment of Fig. 6a marked by two red lines is a possible candidate. By zooming in, see Fig. 6b and c, one can verify that the number of repeating peaks is indeed 256.

By measuring the distance between the peaks, we can find that the processing of one bit by `poly_A2A()` takes 51 points. This parameter is referred to as *bit_offset* in the sequel. Since for `poly_A2A()` the shares $A[i]$ and $R[i]$ are processed immediately following each other (see line 4 of `poly_A2A()` in Fig. 3), *bit_offset* contains both shares.

By locating the first peak, we can find the starting point of `poly_A2A()` procedure. This parameter is referred to as *offset*. Note that we do not need to know neither the value of a random mask, nor the shuffling order to compute the *offset* and *bit_offset*.

7 Profiling stage

The aim of profiling is to construct a neural network model capable of distinguishing between the message bit values "0" and "1." At the attack stage, we use this model to count the number of "1"s in the message in order to determine its HW.

We use neural networks with a multilayer perceptron (MLP) architecture shown in Table 2. It is the same as the one in [40] except for the input size. This architecture was selected using the grid search algorithm [23] which trains a model for every joint specification of hyperparameter values in the Cartesian product of the set of values for each individual hyperparameter.

During training, we use *Nadam* optimizer[19], which is an extension of RMSprop with Nesterov momentum, with

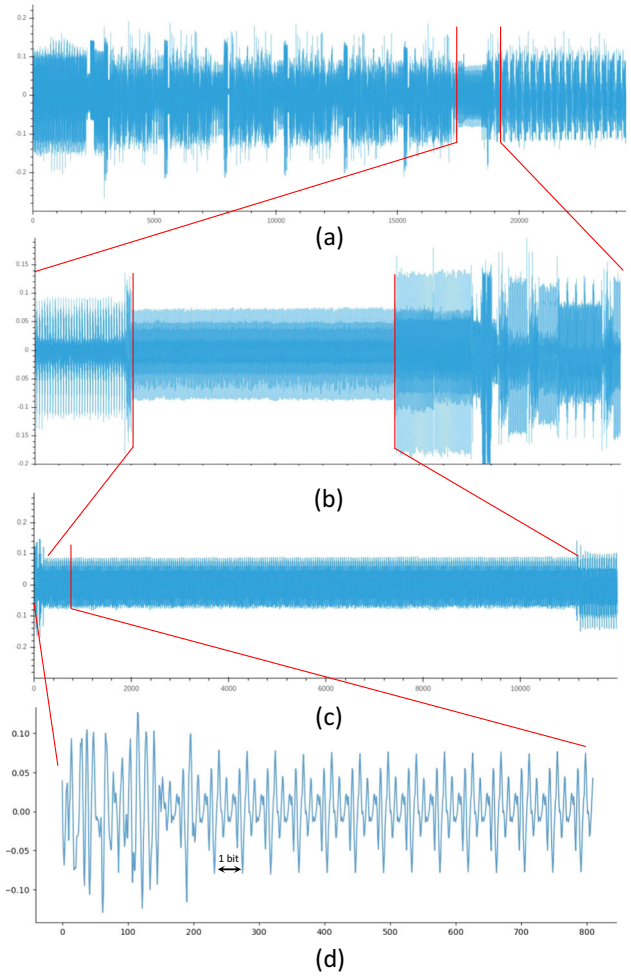


Fig. 6 **a** A power trace representing the execution of the first step of Saber.KEM.Decaps() (average of 50K measurements sampled with decimation 15); **b** An interval containing `poly_A2A(v1,v2)`, `POL2MSG(v1,m1)` and `POL2MSG(v2,m2)`; **c** A detailed view of `poly_A2A(v1,v2)` (sampled with decimation 1); **d** The first 15 bits of `poly_A2A(v1,v2)`

a learning rate of 0.001 and a numerical stability constant epsilon=1e-08. Binary cross-entropy is used as a loss function. The training is run for a maximum of 100 epochs, with a batch size of 128 and an early stopping. 70% of the training set is used for training, and 30% is used for validation.

Unlike [40] where eight models were trained, one for each bit position of a byte, we train a single model capable of recovering all message bits. This is accomplished by composing the training set as a union of trace intervals corresponding to individual bit processing. As a result, we get a universal model which has "learned" features for all 256 bits. Using a cut-and-join technique like this, we can increase the size of the training set by a factor of 256 without having to capture 256 times as many traces. For example, the 2 M training set used in our experiments is composed from 7.8K captured traces. On an ARM Cortex-M4 running at 24MHz, it takes

Table 2 The MLP architecture

Layer type	(Input, output) shape	# Parameters
Batch Normalization 1	(90, 90)	360
Dense 1	(90, 128)	11648
Batch Normalization 2	(128, 128)	512
ReLU	(128, 128)	0
Dense 2	(128, 32)	4128
Batch Normalization 3	(32, 32)	128
ReLU	(32, 32)	0
Dense 3	(32, 16)	528
Batch Normalization 4	(16, 16)	64
ReLU	(16, 16)	0
Dense 4	(16, 1)	17
Softmax	(1, 1)	0
Total parameters	17,385	
Trainable parameters	16,853	

less than 17 min to capture the latter and 3 days to capture the former.

The cut-and-join technique is applicable to `poly_A2A()` leakage point because `poly_A2A()` procedure processes all message bits in the same way during their storage in memory. Thus, traces representing the execution of `poly_A2A()` appear identical for all message bits except the first and last, as we can see from Fig 7. Because of the Cortex-M4's three-stage pipeline, the next instruction begins before the previous instruction has finished. As a result, the power consumed during the processing of the first and the last bits differs from the power consumed during the processing of other bits.

Similarly to [40], we defeat masking by training models on traces containing the bits of both shares labeled by the value of the corresponding message bit. Thus, the models are capable of recovering the message bits directly, without explicitly extracting the mask. However, since the message bits are also shuffled in our case, we cannot train on traces captured from the device under attack for random messages, as in [40] because the order of bits (and thus training labels) is unknown. Instead, we train on a combination of traces from the profiling device running an implementation with deactivated shuffling, and traces from the device under attack captured for all-0 and all-1 messages. Obviously, the labels of all bits are the same for all-0 and all-1 messages. In the experimental results section, we show that such a combined strategy helps us minimize the negative effect of device variability on model's classification accuracy. We also show that training on 100% of traces from the device under attack captured for all-0 and all-1 messages is not the best choice because traces of all-0 and all-1 messages do not allow the neural network to

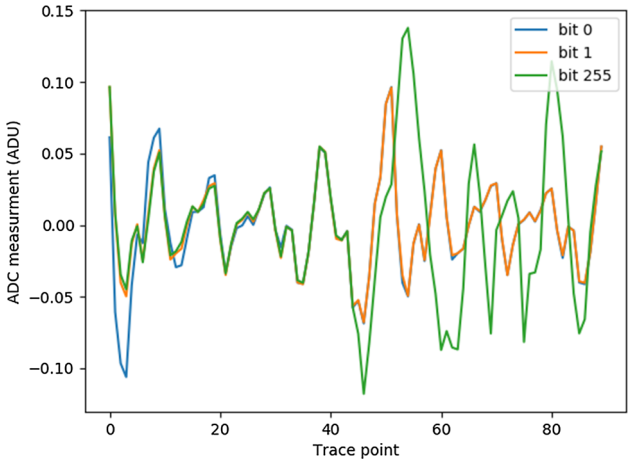


Fig. 7 Average power traces representing the processing of message bits 0, 1 and 255 by `poly_A2A()` (for 10K measurements). Traces for the remaining bits look similarly to the trace of bit 1

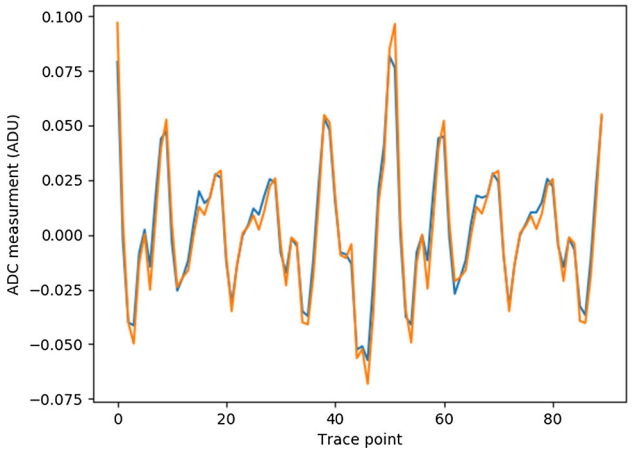


Fig. 8 Comparison of average power traces of D_P and D_A

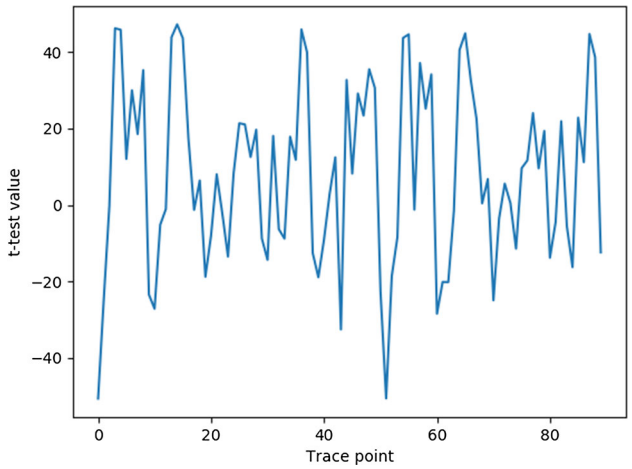


Fig. 9 T-test result for 10K trace sets from D_P and D_A. The peaks represent the points in which the difference between traces is most noticeable

learn all possible features due to the above-mentioned impact of the previous and next instructions on power consumption.

The effect of device variability on the shape of traces is illustrated in Fig. 8. Figure 8 shows two plots obtained by averaging 10K traces captured from the profiling device D_P (blue) and the device under attack D_A (orange) during the processing of the message by `poly_A2A()`. The blue plot is difficult to see because it is covered by the orange plot to a large extent. Figure 9 shows Welch's t-test [61] results for the same 10K trace sets from D_P and D_A computed as:

$$t = \frac{\mu_P - \mu_A}{\sqrt{\frac{\sigma_P^2}{n_P} + \frac{\sigma_A^2}{n_A}}},$$

where μ_P/μ_A, σ_A/σ_P and n_P/n_P are the mean, standard deviation and the size of the trace sets from D_P/D_A. We can see that there are noticeable differences between traces. The bottom peak corresponds to the point 51. Later, we show that side-channel data in the interval around this point is crucial for accurate class prediction.

The pseudo-code of the profiling algorithm is shown in Fig. 10. TrainModel() takes as input the number of traces to be captured, τ, the neural network's input size, *in_size*, and a parameter $k \in \mathbb{I}$, $\mathbb{I} = \{x \in \mathbb{R} \mid 0 \leq x \leq 1\}$, which defines which fraction of traces is captured from the profiling device, D_P. For example, $k = 0.8$ means that 80% of traces are from D_P. The rest of traces is captured from the device under attack, D_A, for all-0 and all-1 messages in equal parts, $r = (1-k)/2$.

At step 1, ComposeTrainingSet() procedure is called to create a set of training traces, $\boldsymbol{\mathcal{T}}$, and the corresponding set of labels, $\boldsymbol{\mathcal{L}}$. In ComposeTrainingSet(), $k \times \tau$ messages are selected at random and encrypted by a fixed public key.[1] The profiling device D_P, which is running an implementation with deactivated shuffling, is used to decapsulate the resulting set of ciphertexts. During its execution, the power traces are captured.

Similarly, $\tau \times r$ all-0 and $\tau \times r$ all-1 messages are generated and encrypted. The device under attack D_A is used to decapsulate the resulting ciphertexts, and the power traces are captured (step 4–8).

Next, the initial offset, *offset*, and the distance between the message bits in $\boldsymbol{\mathcal{T}}'$, *bit_offset*, are determined as described in Sect. 6. Finally, the cut-and-join technique is used to divide $\boldsymbol{\mathcal{T}}'$ into intervals representing individual message bit processing and to generate the set of labels $\boldsymbol{\mathcal{L}}$ containing the corresponding message bit values.

[1] It is also possible to train with different keys. This does not affect the outcome.

TrainModel(τ, *in_size*, k)
1: $(\mathcal{T}, \mathcal{L})$ = ComposeTrainingSet(τ, *in_size*, k)
2: Train $\mathcal{NN} : \mathbb{R}^{in\_size} \rightarrow \mathbb{I}$ on $(\mathcal{T}, \mathcal{L})$
3: **return** $\mathcal{NN}$

ComposeTrainingSet(τ, *in_size*, k)
1: $\boldsymbol{m} = \emptyset, \mathcal{T}' = \emptyset$
2: $\boldsymbol{M} = \{0,1\}^{256}$
3: $(\boldsymbol{m}, \mathcal{T}')$ = CaptureTrace($\boldsymbol{M}, \boldsymbol{m}, \mathcal{T}', D_P, 1, k \times \tau$)
4: $r = (1-k)/2$
5: $\boldsymbol{M} = \{0\}^{256}$
6: $(\boldsymbol{m}, \mathcal{T}')$ = CaptureTrace($\boldsymbol{M}, \boldsymbol{m}, \mathcal{T}', D_A, k \times \tau + 1, r \times \tau$)
7: $\boldsymbol{M} = \{1\}^{256}$
8: $(\boldsymbol{m}, \mathcal{T}')$ = CaptureTrace($\boldsymbol{M}, \boldsymbol{m}, \mathcal{T}', D_A, (k+r) \times \tau + 1, r \times \tau$)
9: Determine initial *offset* and *bit_offset* from $\mathcal{T}'$
10: $\mathcal{T} = \emptyset,\ \mathcal{L} = \emptyset$
11: **for** each $b \in \{0, 1, \ldots, 255\}$ **do**
12: $start = offset + b \times bit\_offset$
13: $stop = start + in\_size$
14: $\mathcal{T} = \mathcal{T} \cup \mathcal{T}'[:, start:stop]$
15: $\mathcal{L} = \mathcal{L} \cup \{\, l(\mathcal{T}_i) \in \{0,1\} \mid l(\mathcal{T}_i) = m_i[b], \forall i \in \{1, \ldots, \tau\}\}$
16: **end for**
17: **return** $(\mathcal{T}, \mathcal{L})$

CaptureTrace($\boldsymbol{M}, \boldsymbol{m}, \mathcal{T}', D, a, b$)
1: **for** each $i \in \{a, a+1, \ldots, a+b\}$ **do**
2: $m_i \leftarrow \mathcal{U}(\boldsymbol{M})$
3: $\boldsymbol{m} = \boldsymbol{m} \cup \{m_i\}$
4: $r_i \leftarrow \mathcal{U}(\{0,1\}^{256})$
5: $pk_i \leftarrow \mathcal{U}(\{0,1\}^{256})$
6: c_i = Saber.PKE.Enc$(pk_i, m_i; r_i)$
7: $\mathcal{T}_i \Leftarrow D$[Saber.KEM.Decaps$(c_i)$]
8: $\mathcal{T}' = \mathcal{T}' \cup \{\mathcal{T}_i\}$
9: **end for**
10: **return** $(\boldsymbol{m}, \mathcal{T}')$

Fig. 10 Profiling algorithm

8 Attack stage

To defeat the combined masked and shuffled countermeasures, we make use of the existing key and message recovery techniques presented in [40] and [45] for masked-only and shuffled-only LWE/LWR-based KEMs, respectively, and introduce two new algorithms.

In this section, we outline the main steps of the proposed secret and session key recovery approaches, then describe the key and message recovery techniques from from [40] and [45], and finally present the new algorithms.

8.1 Secret key recovery

The secret key is recovered as follows:

(1) Construct 24 chosen ciphertexts $c_1, \ldots, c_{24}$ as described in Sect. 8.2.
(2) For each c_i, $i \in \{1, \ldots, 24\}$, construct 256 ciphertexts $c_{i_0}, \ldots, c_{i_{255}}$ such that c_{i_j} decrypts to m'_{i_j} = Saber.PKE.Dec($\mathbf{s}, c_{i_j}$) which is equal to the message m'_i = Saber.PKE.Dec($\mathbf{s}, c_i$) with the jth bit is flipped, for $j \in \{0, \ldots, 255\}$. The procedure is described in Sect. 8.3.
(3) For each of 24×257 resulting ciphertexts, acquire a power trace during the decapsulation of the ciphertext by the device under attack. Repeat N times for each ciphertext.
(4) Use the acquired $24 \times 257 \times N$ power traces to recover the messages m'_i contained in the ciphertexts c_i, for all $i \in \{1, \ldots, 24\}$, using RecoverMessage() algorithm presented in Sect. 8.4.
(5) Derive the secret key from the 24 recovered messages $m'_1, \ldots, m'_{24}$ as described in Sect. 8.2.

Session key recovery Assume that the adversary has a properly generated ciphertext c which is decapsulated by the device under attack. The adversary follows the steps (2)–(5) of the secret key recovery algorithm described above to extract the message m' contained in c from $257 \times N$ power traces. Given m', he/she computes $(\hat{K}', r') = \mathcal{G}(pkh, m')$ and gets the session key as $K = \mathcal{H}(\hat{K}', c)$.

8.2 Chosen ciphertext construction

In [40], an approach based on error-correcting codes (ECC) was introduced to recover the secret key from masked Saber. We use the same chosen ciphertexts as in [40] for recovering the secret key from masked and shuffled Saber.

The ciphertexts are constructed as $c_j = (\boldsymbol{c_m}, \mathbf{b}')$ where $\boldsymbol{c_m} = k_0 \sum_{i=0}^{255} x^i \in R_T$ and

$$\mathbf{b}' = \begin{cases} (k_1, 0, 0) \in R_p^{3\times 1} \text{for} \quad j = \{1, \ldots, 8\}, \\ (0, k_1, 0) \in R_p^{3\times 1} \text{for} \quad j = \{9, \ldots, 16\}, \\ (0, 0, k_1) \in R_p^{3\times 1} \text{for} \quad j = \{17, \ldots, 24\}, \end{cases}$$

where the pairs (k_0, k_1) are listed in Table 3. In this table, the ith coefficient of the secret key $\mathbf{s}$, $\mathbf{s}[i]$, is mapped into a codeword of the $[8, 4, 4]_2$ extended Hamming code composed from the eight message bits. The first 256 secret key coefficients are derived from messages recovered from $c_1, \ldots, c_8$, the second 256 coefficients—from $c_9, \ldots, c_{16}$ and the last 256 coefficients—from $c_{17}, \ldots, c_{24}$.

The approach in [40] works because decryption of $(\boldsymbol{c_m}, \mathbf{b}')$ yields the message

$$m' = ((\mathbf{b}'^T(\mathbf{s} \mod p) + h_2 - 2^{\epsilon_p - \epsilon_T}\boldsymbol{c_m}) \mod p) \gg (\epsilon_p - 1) \in R_2,$$

whose ith bit, $m'[i]$, is a function of the triple $(k_0, k_1, \mathbf{s}[i])$:

$$m'[i] = ((k_1 \cdot (\mathbf{s}[i] \mod p) + H - 2^{\epsilon_p - \epsilon_T}k_0) \mod p) \gg (\epsilon_p - 1), \quad (1)$$

Table 3 Pairs (k_1, k_0) which are used to derive secret key coefficients $\mathbf{s}[i]$ from eight message bits [40]

$\mathbf{s}[i]$	The message bit value for the pair (k_1, k_0)							
	(186,0)	(293,7)	(311,7)	(615,2)	(613,2)	(890,4)	(903,4)	(199,0)
−4	0	1	1	1	1	0	0	0
−3	1	1	1	0	0	0	0	1
−2	1	0	0	1	1	0	0	1
−1	0	0	0	0	0	0	0	0
0	0	1	1	0	0	1	1	0
1	0	0	0	1	1	1	1	0
2	1	0	0	0	0	1	1	1
3	1	1	1	1	1	1	1	1
4	1	1	0	1	0	0	1	0

where $H = 2^{\epsilon_p - 2} - 2^{\epsilon_p - \epsilon_T - 1} + 2^{\epsilon_q - \epsilon_p - 1}$. Thus, $m'[i]$ leaks information about $\mathbf{s}[i]$.

8.3 Bit-flip technique

In [45], a technique called *bit-flip* was introduced to recover the message m' contained in ciphertext c which is decapsulated by the device under attack implementing a shuffled LWE/LWR-based KEM algorithm. We use a "fuzzy" version of this technique, presented in Sect. 8.4, for recovering messages contained in 24 chosen ciphertexts which are decapsulated by the device under attack implementing masked and shuffled Saber.

Given a ciphertext $c = (\boldsymbol{c_m}, \mathbf{b}')$, the bit-flip technique [45] constructs 256 ciphertexts c_j, $j \in \{0, \ldots, 255\}$, in which the value of the center of the integer ring $\mathbb{Z}_q$ is subtracted from the jth coefficient of $\boldsymbol{c_m}$. Since the message polynomial is only additively hidden within the ciphertext, this results in a ciphertext decrypting m'_j which is equal $m' =$ Saber.PKE.Dec($\mathbf{s}$, c) with jth bit flipped.

For c and each c_j, a side-channel HW classifier is applied to find the HW of m' and each m'_j, for $j \in \{0, \ldots, 255\}$. In [45], the HW classifier is constructed by the template approach. From the obtained HWs, the message m' is recovered bit-by-bit as follows:

$$m'[i] = \begin{cases} 0 \text{ if } \mathrm{HW}(m'_j[i]) = \mathrm{HW}(m'[i]) + 1 \\ 1 \text{ if } \mathrm{HW}(m'_j[i]) = \mathrm{HW}(m'[i]) - 1. \end{cases} \quad (2)$$

8.4 Message HW recovery algorithm

In this section, we present the algorithm RecoverHW() which we use to recover HW of messages contained in 24 chosen ciphertexts and their bit-flipped versions. Its pseudo-code is shown in Fig. 11.

RecoverHW() takes as input the neural network trained at the profiling stage, $\mathcal{NN}$, the neural network's input size, *in_size*, the initial offset, *offset*, the distance between the bits, *bit_offset*, the ciphertext c for which the message HW has to be recovered and the degree of repetition of the same measurement, N.

```
RecoverHW(NN,in_size, offset, bit_offset, c, N)
 1: for each i ∈ {1, ..., N} do
 2:     T̂_i ⇐ D_A[Saber.KEM.Decaps(c)]
 3:     HW_i = 0
 4:     for each b ∈ {0, ..., 255} do
 5:         start = offset + b × bit_offset
 6:         stop = start + in_size
 7:         s_b = NN(T̂[start:stop])
 8:         if s_b > 0.5 then
 9:             HW_i = HW_i + 1
10:         end if
11:     end for
12: end for
13: N' = RemoveOutliers(HW_1, ..., HW_N)
14: HW = Median(HW_1, ..., HW_N')
15: return HW
```

Fig. 11 Message HW recovery algorithm

First, N trials are performed to recover the HW of the message m' contained in c. The device under attack is used to decapsulate c, and a power trace $\hat{\mathcal{T}}_i$ is captured during its execution (step 2). The interval corresponding to the processing of b in $\hat{\mathcal{T}}_i$ is located based on *offset* and *bit_offset* for each of the 256-bit positions $b \in \{0, 1, \ldots, 255\}$ (representing the message bits in an unknown shuffled order) (steps 5–6). This interval is fed into the neural network $\mathcal{NN}$ trained during the profiling stage to determine whether the message bit in position b has a value of "0" or "1." If the resulting score s_b is greater than 0.5 (i.e., "1" has a higher probability), the HW is incremented. Otherwise, the HW is not changed.

The HW is then determined by first removing the outliers and then computing the median of the remaining HWs (steps 13–14). An outlier is defined as a HW that differs from the median HW by more than 10%. We explored a variety of combining methods. The one we present consistently outperforms others in our experiments.

```
RecoverMessage(NN,in_size, offset, bit_offset, c, N)
1:  HW(m') = RecoverHW(NN,in_size, offset, bit_offset, c, N)
2:  for each i ∈ {0, ..., 255} do
3:      c_i = BitFlip(c, i)
4:      HW(m'_i) = RecoverHW(NN,in_size, offset, bit_offset, c_i, N)
5:      if HW(m'_i) > HW(m') then
6:          m'[i] = 0
7:      else
8:          if HW(m'_i) < HW(m') then
9:              m'[i] = 1
10:         else
11:             m'[i] = 2 /* error detected */
12:         end if
13:     end if
14: end for
15: return m'
```

Fig. 12 Message recovery algorithm

8.5 Message recovery algorithm

In this section, we present a "fuzzy" version of the bit-flip technique, RecoverMessage(). We construct 256 ciphertexts containing bit-flipped messages in the same way as in the original method [45]. However, we take a different approach to deciding the final message bit values. We also quantify the success rate of message HW recovery as a function of the success rate of single-bit recovery.

The pseudo-code is shown in Fig. 12. RecoverMessage() takes as input the same parameters as RecoverHW() algorithm. First, the HW of m' contained in c is recovered by calling RecoverHW(). Then, the following loop is repeated 256 times: For each $i \in \{0, \ldots, 255\}$, the ciphertext c_i is constructed using BitFlip(), i.e., the value of the center of $\mathbb{Z}_q$ is subtracted from the ith coefficient of c. The HW of m'_i contained in c_i is recovered by calling RecoverHW(). If $\mathrm{HW}(m'_i) > \mathrm{HW}(m')$, the ith bit of m' is assigned "0." If $\mathrm{HW}(m'_i) < \mathrm{HW}(m')$, the ith bit of m' is assigned "1." Otherwise the ith bit of m' is assigned '2' to indicate that the bit is not recovered correctly. In the experiments, we call this case a *detectable error*.

Next we quantify the probability to recover the message HW as a function of the probability to recover the single message bit. The property below assumes that the message is balanced, i.e., has equal number of "1"s and "0"s.

Property 1 *Let m be a balanced n-bit binary message. If p is the success rate of single-bit recovery and bit errors are mutually independent events, then the success rate of message HW recovery is given by:*

$$p_{HW} = \sum_{i=0}^{n/2} \binom{n/2}{i}^2 p^{n-2i}(1-p)^{2i} \tag{3}$$

Table 4 Success rate of 256-bit message HW recovery

p	p_{HW}
0.99	0.279883
0.999	0.879911
0.9999	0.987280
0.99999	0.998719

Proof The proof is based on the fact that, if, for any $0 \le k \le n/2$, k message bits change as $0 \to 1$ and other k message bits change as $1 \to 0$, then the message HW does not change.

A n-bit balanced binary message has $n/2$ "0"s and $n/2$ "1"s. There are $\binom{n/2}{k}$ choices to select k elements from a set of size $n/2$. Thus, for a fixed k, the number of possible $2k$-bit errors in which k bits flip in one direction and the rest of bits flip in another direction is $\binom{n/2}{k}^2$. Since the probability of a $2k$-bit error in an n-bit message is $p^{n-2k}(1-p)^{2k}$, we get (3).

Using Property 1, we can estimate the success rate of single-bit recovery required to recover the message HW. Table 4 lists some examples. According to the table, the success rate of single-bit recovery should be of the order of 0.999 to recover the message HW with a high probability.

9 Experimental results

In the experiments, we use two identical CW303 ARM devices, D_P and D_A. D_P is the profiling device. We have complete control over D_P, which means we can reload it with a different implementation, change its secret key, etc. D_A is the device that is being attacked. We use D_A to capture traces for key recovery and a part of traces for training.

9.1 Message recovery

In this section, we evaluate the impact of training set composition on the success rate of RecoverMessage() algorithm. We also justify why the use of an ensemble of a set of models can further improve the success rate.

We trained MLP models on trace sets of size 2 M with varying proportions of D_P and D_A traces, denoted by $D_P : D_A$. We tried five cases: $D_P : D_A = \{0{:}100, 20{:}80, 50{:}50, 80{:}20, 100{:}0\}$. The notation $x : y$ means that $x\%$ of traces are from D_P and $y\%$ are from D_A. Recall from Sect. 7 that traces from D_P are captured for random messages, while traces from D_A are captured for all-0 and all-1 messages in equal proportion. D_P runs an implementation with deactivated shuffling.

Table 5 The impact of training set composition on message recovery success rate

N	Training set	# Detected (d) and undetected (u) errors in the test set $i \in \{0, \ldots, 9\}$																				Average	
		0		1		2		3		4		5		6		7		8		9			
	$D_P : D_A$	d	u	d	u	d	u	d	u	d	u	d	u	d	u	d	u	d	u	d	u	d	u
	0:100	1	1	0	0	0	0	0	0	0	1	0	0	0	1	0	1	0	0	0	0	0.1	0.4
	20:80	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	2	0	0.3	0.2
20	50:50	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	0.2	0.1
	80:20	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	100:0	2	5	0	2	0	2	4	3	0	0	0	3	0	7	3	0	3	5	2	7	1.4	3.4
	0:100	1	0	0	0	0	2	0	2	0	3	1	2	0	2	2	1	0	0	0	2	0.4	1.3
	20:80	1	0	0	0	1	1	0	1	0	0	0	0	1	1	1	1	0	0	1	0	0.5	0.4
15	50:50	3	2	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	1	2	0.3	0.4
	80:20	0	0	0	0	0	1	0	1	0	0	0	0	0	0	1	0	0	0	1	1	0.2	0.3
	100:0	3	8	1	3	1	1	1	6	0	1	1	3	0	8	2	1	2	7	4	13	1.5	5.1
	0:100	2	3	1	1	0	3	3	2	8	3	1	1	2	5	3	2	0	1	0	3	2.0	2.4
	20:80	4	6	0	1	0	1	1	0	1	4	0	0	1	4	0	4	0	0	1	5	0.8	2.5
10	50:50	5	3	0	1	0	1	0	0	1	2	0	0	1	0	0	2	0	0	3	8	1.0	1.7
	80:20	1	4	0	0	1	2	0	1	1	0	0	0	0	1	0	2	0	0	4	2	0.7	1.2
	100:0	3	18	3	4	3	7	3	8	6	5	3	6	5	7	1	4	4	9	3	14	3.4	8.2

For each fraction $D_P : D_A = x : y$, we trained ten models with the architecture in Table 2 using TrainModel() with input parameters $\tau = 2\,\text{M}$ and $k = x/100$ and selected the best.

We tested the models on ten different ciphertexts created by encrypting a random message with a randomly selected public key. To recover the message, $257 \times N = 5140$ traces from D_A were captured for each ciphertext, for $N = 10$, 15 and 20.

Table 5 lists the number of detected and undetected errors for each of the ten test sets. Recall that detected errors are those for which RecoverMessage() returns "2" as the message bit value. The ability to detect errors is very useful since e detected bit errors can be handled by enumerating 2^e possible choices, computing $(\hat{K}', r') = \mathcal{G}(pkh, m')$ and then checking if $c = \text{Saber.PKE.Enc}(pk, m'; r')$.

We can see from Table 5 that the model trained on a combination of 80% of traces from D_P and 20% of traces from D_A produces the best results. Including traces from the device under attack into the training set helps mitigating the negative effect of device variability on classification accuracy.

We can also see that training on 100% of traces from the device under attack captured for all-0 and all-1 messages is not the best choice. As we mentioned in Sect. 7, due to the Cortex-M4's three-stage pipeline, the power consumed during the processing of a given message bit depends not only on the value of that bit, but also on values of previous bits. Therefore, traces of all-0 and all-1 messages do not allow the neural network to learn all possible features.

Training on 100% of traces from the profiling device is the worst option. One could argue that such an option has the advantage of allowing profiling to be completed prior to the attack. However, thanks to the cut-and-join technique, we only need to capture 1.5K traces from D_A to contribute 20% of traces to the 2 M training set, which takes less than 4 min. As a result, composing the training set as 80:20 has no significant effect on the time required to physically access D_A.

Table 5 also shows that, for $N = 15$ and lower, all models have some undetected errors. It is possible to improve the success rate by increasing the value of N, however, a larger N increases capture time for attack traces, which is undesirable. Thus, in the experiments that follow, rather than increasing N, we use an ensemble of a set of k models to improve the success rate of message recovery. The ensemble approach increases training time, but this is not as critical as increasing access time to D_A.

It is known [23] that, on average, an ensemble of a set of models performs at least as well as any of its members. Furthermore, if the members make independent errors, then the ensemble performs considerably better than its members. This can be justified as follows. Suppose that each model makes an error ϵ_i on each test example, and the errors are drawn from a zero-mean multivariate normal distribution with variances $\mathbb{E}[\epsilon_i^2] = v$ and covariances $\mathbb{E}[\epsilon_i \epsilon_j] = c$. Then, the error made by the average prediction of all the ensemble models is $\frac{1}{k}\sum_i \epsilon_i$. So, the expected squared error of the ensemble predictor is given by [23]:

Table 6 Success rate of key recovery (average for 10 tests)

N	k	# Errors		Attack time		
		d	u	Capture	Message rec.	Key enum.
	5	0	0		23.10 min	0 s
20	3	2.6	0	5.6 h	9.15 min	3.66 s
	1	76.6	8.3		3.05 min	–
	5	0.2	0		11.34 min	0.02 s
15	3	2.5	0	4.2 h	7.15 min	2.93 s
	1	94.4	12.6		2.20 min	–
	5	1.2	0		8.12 min	0.17 s
10	3	9.3	0	3.2 h	4.89 min	104.69 days
	1	95.0	16.7		1.68 min	–

$$\mathbb{E}\left[\left(\frac{1}{k}\sum_i \epsilon_i\right)^2\right] = \frac{1}{k^2}\,\mathbb{E}\left[\sum_i\left(\epsilon_i^2 + \sum_{i\neq j}\epsilon_i\epsilon_j\right)\right] = \frac{v}{k} + \frac{(k-1)c}{k}.$$

From the above we can conclude that, if the errors are dependent and $c = v$, then the expected squared error of the ensemble is v, i.e., the ensemble brings no improvement. In contract, if the errors are independent and $c = 0$, then the expected squared error of the ensemble reduces to $\frac{v}{k}$, i.e., it is inversely proportional to the ensemble size k.

The ensemble approach helps us in practice because different models typically do not make all the same errors on the test set, as we can see from the results in Table 5. This might due to differences in models parameters after training. In Sect. 9.3, we present an example illustrating these differences.

9.2 Secret key recovery

To evaluate the success rate of the secret key recovery attack, we captured ten test sets of $24 \times 257 \times N$ traces representing the decapsulation of ciphertexts constructed following steps 1–3 of the procedure in Sect. 8.1, for $N = 10, 15$ and 20. Each test set was captured for a different secret key.

To recover the secret messages contained in the ciphertexts, we use an ensemble of best models obtained during training. The ensemble method is known to be useful in side-channel analysis [43, 57]. Table 6 shows the results for ensembles of size up to 5 for different N. The k models in the ensemble are trained on the same training set with $D_P : D_A = 80 : 20$.

The output of an ensemble of k models is obtained as follows. For each $j \in \{0, 1, \ldots, 255\}$, models that result in $m'[j] = 2$ (i.e., detected error) are excluded from voting, and then the mean of the $m'[j]$s produced by the remaining models is computed. If the mean is ≤ 0.5, the jth message bit is set to "0"; otherwise it is set to "1." Finally, the secret key is derived from the 24 recovered messages as described in Sect. 8.2.

Since we use the ECC-based method [40] which is able to correct single errors and detects one additional error in the recovered message, we can mark the positions of the detected incorrect key coefficients for later enumeration. With d detected incorrect key coefficients, 9^d enumerations are required to find the true key. For example, for $N = 10$ and $k = 5$, $9^{1.2} \approx 14$ enumerations are required.

Undetected errors are positions that are not handled by the ECC. We determine them by comparing the recovered key to the true key. One can see from Table 6 that, for $N \geq 10$ and $k \geq 3$, there are no undetected errors. Certainly, the values of N and k may vary depending on the implementation, environmental conditions, acquisition method, etc.

The last three columns of Table 6 show the time required for capturing traces and message recovery, as well as the average key enumeration time on a PC with a 16 core processor running at 4.3 GHz and 64 GB of RAM (simple single threaded implementation). The sign "-" means that key enumeration is not feasible. Note that capture requires physical access to the device under attack, whereas post-processing steps do not.

9.3 Analysis of neural network models

It is challenging to explain how neural network models take their decisions. However, making an attempt is important because it could help in locating and fixing vulnerabilities in the implementation under attack. It might also aid in model optimization.

9.3.1 Feature analysis

To assess the significance of various input features for the models, we use two techniques:

(1) weight analysis, and
(2) stuck-at-0 fault injection.

Both methods have been shown effective in previous attacks of lattice-based PKE/KEMs [60].

Figure 13c shows the gamma, γ, parameters of the input Batch Normalization layer of five MLP models in Table 5 after training. The model trained on the dataset composed as $D_P : D_A = x : y$ is referred to as DP_x_DA_y.h5 in the legend. Recall that Batch Normalization first standardizes the input values X of the layer using their respective mean, μ, and standard deviation, σ, $X_{\text{norm}} = (X - \mu)/\sigma$, and then applies the scaling, γ (gamma) and offset, β (beta), parameters to the result, $X' = (\gamma * X_{\text{norm}}) + \beta$. The parameters γ

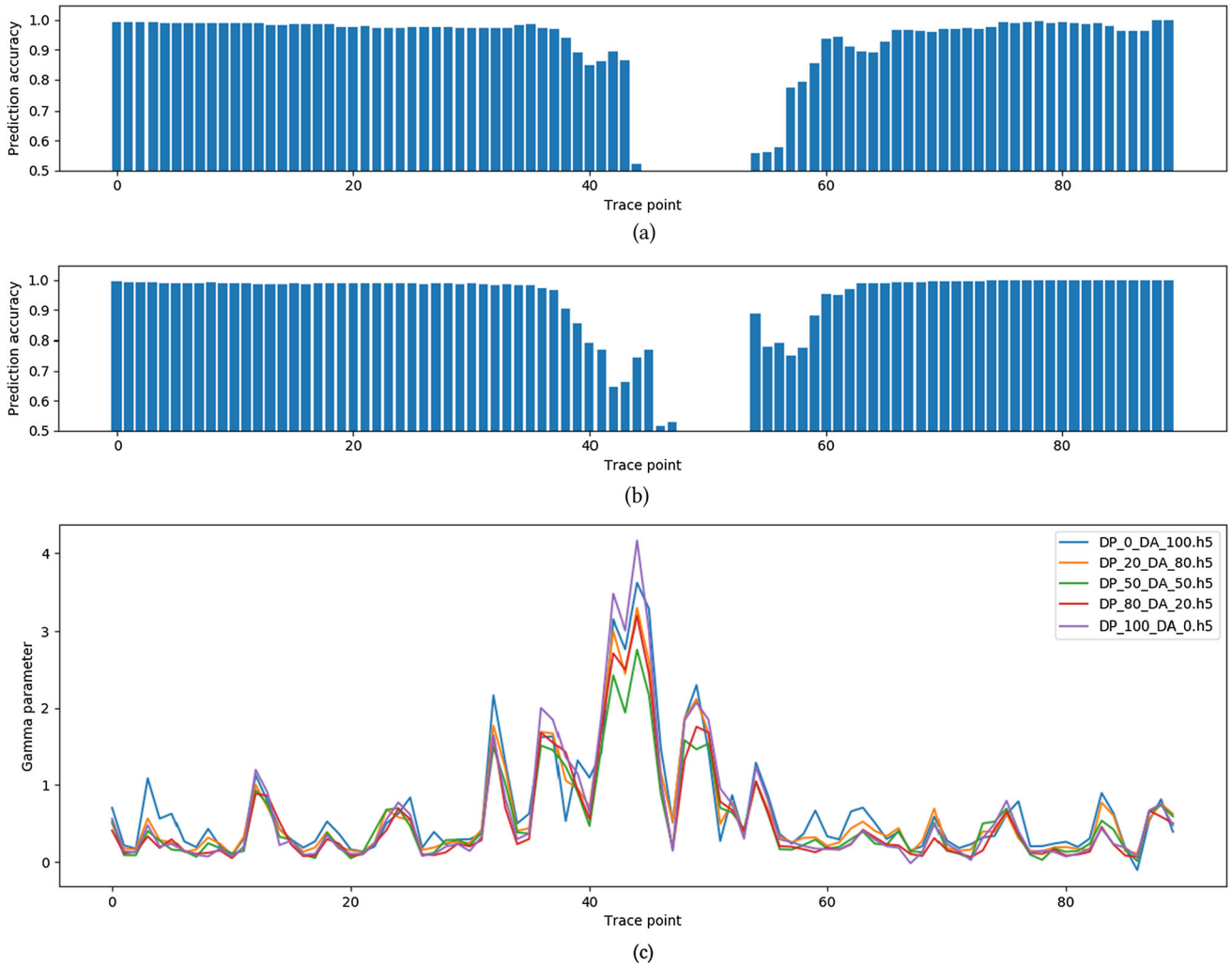


Fig. 13 **a** and **b** The average prediction accuracy of the models DP_80_DA_20.h5 and DP_100_DA_0.h5, respectively, for the case when a given data point is stuck to 0; **c** Gamma parameters of the input Batch Normalization layer of five models

and β are learned by the model during the training process. More specifically, the backpropagation algorithm is adjusted to operate on the transformed inputs, and error is used to update the new scaling and offset parameters learned by the model. Thus, a higher value of γ indicates the higher importance of the corresponding input feature in the decision taken by the model. We can see that there are substantial differences in the weights of different trace points.

The contribution of each feature becomes even more clear after the stuck-at-0 fault injection analysis. Figure 13a and b shows how the prediction accuracy of the models DP_80_DA_20.h5 and DP_100_DA_0.h5, respectively, is affected by setting each single point p of a test trace to 0 before making inference (implying that the model takes its decision without the data sampled at that point). If the prediction accuracy drops to the random guess accuracy of 0.5, the point p is important.

In Fig. 13a and b, we can see that there are points in the interval [45:53] whose removal drops the accuracy to the random guess. It shows that the most important input features for the model's decision are located there. This, in turn, implies that the computations performed by the implementation of Saber during the corresponding clock cycles leak exploitable side-channel information. By doing a clock cycle accurate analysis of the assembly code of poly_A2A_shuffled() in Fig. 4b, one can link these computations to the store register halfword instructions strh.w r3, [r4, r1, lsl 1] and strh.w r2, [r5, r1, lsl 1] which store a halfword from a register to memory. These instruction implement the line 6 the C code of poly_A2A_shuffled() marked in red in Fig. 3.

9.3.2 Model comparison

Figure 13 also illustrates that different models can be non-equally "sensitive" to the same point of data. This may result

in these models making different inference errors on the same test set.

By comparing Fig. 13a and b, we can see that the deletion of some points may affect the models DP_80_DA_20.h5 and DP_100_DA_0.h5, differently. For example, the deletion of the point 42 decreases the prediction accuracy of the model DP_80_DA_20.h5 to 87% only, while for the model DP_100_DA_0.h5 the reduction is to 65%. Contrary, the deletion of the point 54 drops the prediction accuracy of the model DP_80_DA_20.h5 to 56% while the accuracy of the model DP_100_DA_0.h5 reduces only to 89%. While all models follow a similar "pattern" of weights in Fig. 13c, for some data points their values differ. This also applies to the parameters of the follow up layers, causing the avalanche effect. As a result, the same data point might contribute non-equally to the decisions of different models. According to Table 5, the model DP_80_DA_20.h5 is more successful than DP_100_DA_0.h5 in making correct predictions. Thus, apparently the weights learned by DP_80_DA_20.h5 are closer to the optimal than the ones learned by DP_100_DA_0.h5.

10 Conclusion

We demonstrated that it is possible to break a masked and shuffled implementation of Saber KEM using deep learning-based power analysis. Earlier it was believed that the masked and shuffled countermeasures, when combined, provide adequate protection against side-channel attacks. The presented message and key recovery attacks are not specific to Saber and might be applicable to other LWE/LWR-based PKE/KEMs, including CRYSTALS-Kyber [4] which has been recently selected for standardization by NIST [36].

The traces, models and scripts, along with video demonstration of the attack are publicly available at https://drive.google.com/drive/folders/1NBf1oLO81UTSf_Z4HRRScbRMIRzRNzc

Future work includes designing stronger countermeasures for LWE/LWR-based PKE/KEMs.

Acknowledgements This work was supported in part by the Swedish Civil Contingencies Agency (Grants No. 2020-11632), the Swedish Research Council (Grants No. 2018-04482 and 2019-04166) and the Swedish Foundation for Strategic Research (Grant No. RIT17-0005).

Funding Open access funding provided by Royal Institute of Technology.

References

1. Agrawal, D., Archambeault, B., Rao, J.R., Rohatgi, P.: The EM side-channel(s). In: Cryptographic Hardware and Embedded Systems, pp. 29–45 (2003)
2. Amiet, D., Curiger, A., Leuenberger, L., Zbinden, P.:Defeating NewHope with a single trace. In: International Conference on Post-Quantum Cryptography, pp. 189–205. Springer (2020). https://doi.org/10.1007/978-3-030-44223-1_11
3. Archambeau, C., Peeters, E., Standaert, F.X., Quisquater, J.J.: Template attacks in principal subspaces. In: Cryptographic Hardware and Embedded Systems, pp. 1–14 (2006)
4. Avanzi, R.M., Bos, J.W., Ducas, L., Kiltz, E., Lepoint, T., Lyubashevsky, V., Schanck, J.M., Schwabe, P., Seiler, G., Stehlé, D.: CRYSTALS-Kyber algorithm specifications and supporting documentation (2020)
5. Barthe, G., Belaïd, S., Espitau, T., Fouque, P.-A., Grégoire, B., Rossi, M., Tibouchi, M.: Masking the GLP lattice-based signature scheme at any order. In: Nielsen, J.B., Rijmen, V. (eds.) Advances in Cryptology - EUROCRYPT 2018, pp. 354–384. Springer International Publishing, Cham (2018)
6. Beirendonck, M.V., D'Anvers, J-P., Karmakar, A., Balasch, J., Verbauwhede, I.: A side-channel resistant implementation of SABER. Cryptology ePrint Archive, Report 2020/733 (2020). https://eprint.iacr.org/2020/733
7. Belleville, N., Courousse, D., Heydemann, K., Charles, H.-P.: Automated software protection for the masses against side-channel attacks. ACM Trans. Archit. Code Optim. **16**(4), 1 (2018)
8. Bhasin, S., D'Anvers, J-P., Heinz, D., Pöppelmann, T., Van Beirendonck, M.: Attacking and defending masked polynomial comparison for lattice-based cryptography. Cryptology ePrint Archive, Paper 2021/104 (2021). https://eprint.iacr.org/2021/104
9. Bhasin, S., D'Anvers, J-P., Heinz, D., Pöppelmann, T., Van Beirendonck, M.: Attacking and defending masked polynomial comparison for lattice-based cryptography. IACR Trans. Cryptogr. Hardw. Embed. Syst. **3**, 334–359 (2021). https://doi.org/10.46586/tches.v2021.i3.334-359
10. Brisfors, M., Forsmark, S., Dubrova, E.: How deep learning helps compromising USIM. In: Proceedings of the 19th Smart Card Research and Advanced Application Conference (CARDIS'2020) (2020)
11. Brumley, B.B., Hakala, R.M., Nyberg, K., Sovio, S.: Consecutive S-box lookups: a timing attack on SNOW 3G. In: Soriano, M., Qing, S., López, J. (eds.) Information and Communications Security, pp. 171–185. Springer, Berlin, Heidelberg (2010)
12. Cagli, E, Dumas, C., Prouff, E.: Convolutional neural networks with data augmentation against jitter-based countermeasures. In: Cryptographic Hardware and Embedded Systems - CHES 2017, pp. 45–68 (2017)
13. Camurati, G., Poeplau, S., Muench, M., Hayes, T., Francillon, A.: Screaming channels: when electromagnetic side channels meet radio transceivers. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, pp. 163–177 (2018)
14. Chari, S., Jutla, C.S., Rao, J.R., Rohatgi, P.: Towards sound approaches to counteract power-analysis attacks. In: Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology

Conference, USA, vol. 1666, pp. 398–412. Springer (1999). https://doi.org/10.1007/3-540-48405-1_26
15. Chen, C., Danba, O., Stein, J., Hülsing, A., Rijneveld, J., Schanck, J.M., Schwabe, P., Whyte, W., Zhang, Z.: NTRU algorithm specifications and supporting documentation (2020). https://csrc.nist.gov/projects/postquantum-cryptography/round-3-submissions
16. Coron, J., Kizhvatov, I.: An efficient method for random delay generation in embedded software. In: Clavier, C., Gaj, K. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2009, pp. 156–170. Springer, Berlin, Heidelberg (2009)
17. CW308 UFO Target. [n.d.]. https://wiki.newae.com/CW308_UFO_Target. Accessed 2022
18. D'Anvers, J., et al.: SABER algorithm specifications and supporting documentation (2020). https://csrc.nist.gov/projects/postquantum-cryptography/round-3-submissions
19. Dozat, T.: Incorporating nesterov momentum into adam (2016)
20. Durstenfeld, R.: Algorithm 235: random permutation. Commun. ACM **7**(7), 420 (1964). https://doi.org/10.1145/364520.364540
21. Fujisaki, E., Okamoto, T.: Secure integration of asymmetric and symmetric encryption schemes. J. Cryptol. **26**(1), 80–101 (2013). https://doi.org/10.1007/s00145-011-9114-1
22. Gérard, F., Rossi, M.: An efficient and provable masked implementation of qTESLA. In: International Conference on Smart Card Research and Advanced Applications, pp. 74–91. Springer (2019)
23. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016). http://www.deeplearningbook.org
24. Goodwill, G., Jun, B., Jaffe, J., Rohatgi, P.: A testing methodology for side-channel resistance validation. In: NIST Non-Invasive Attack Testing Workshop, vol. 7, pp. 115–136 (2011)
25. Guo, Q., Johansson, T., Nilsson, A.: A key-recovery timing attack on post-quantum primitives using the Fujisaki-Okamoto transformation and its application on FrodoKEM. In: Micciancio, D., Ristenpart, T. (eds.) Advances in Cryptology - CRYPTO 2020, pp. 359–386. Springer International Publishing, Cham (2020)
26. Guo, Q., Johansson, T., Nilsson, A.: A key-recovery timing attack on post-quantum primitives using the Fujisaki-Okamoto transformation and its application on FrodoKEM. In: Advances in Cryptology - CRYPTO 2020: 40th Annual International Cryptology Conference, CRYPTO 2020, Santa Barbara, CA, USA, August 17–21, 2020, Proceedings, Part II. Santa Barbara, CA, USA, pp. 359-386. Springer, Berlin, Heidelberg (2020). https://doi.org/10.1007/978-3-030-56880-1_13
27. Hoffman, C., Gebotys, C., Aranha, D.F., Cortes, M., Araújo, G.: Circumventing uniqueness of XOR Arbiter PUFs. In: 2019 22nd Euromicro Conference on Digital System Design (DSD), pp. 222–229 (2019). https://doi.org/10.1109/DSD.2019.00041
28. Hofheinz, D., Hövelmanns, K., Kiltz, E.: A modular analysis of the Fujisaki-Okamoto transformation. In: Theory of Cryptography: 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12–15, 2017, Proceedings, Part I, Baltimore, USA, pp. 341-371 Springer, Berlin, Heidelberg (2017). https://doi.org/10.1007/978-3-319-70500-2_12
29. Kim, J., Picek, S., Heuser, A., Bhasin, S., Hanjalic, A.: Make some noise. Unleashing the power of convolutional neural networks for profiled side-channel analysis. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2019**(3), 148–179 (2019)
30. Kocher, P., Jaffe, J., Jun, B.: Differential power analysis. In: Annual International Cryptology Conference, pp. 388–397. Springer (1999)
31. Kocher, P.C.: Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz, N. (ed.) Advances in Cryptology — CRYPTO '96, pp. 104–113. Springer, Berlin, Heidelberg (1996)
32. Maghrebi, H., Portigliatti, T., Prouff, E.: Breaking cryptographic implementations using deep learning techniques. In: Security, Privacy, and Applied Cryptography Engineering. Springer International Publishing (2016)
33. Maghrebi, H., Servant, V., Bringer, J.: There is wisdom in harnessing the strengths of your enemy: customized encoding to thwart side-channel attacks. In: Peyrin, T. (ed.) Fast Software Encryption, pp. 223–243. Springer, Berlin, Heidelberg (2016)
34. Masure, Loïc., Belleville, N., Cagli, E., Cornelie, M-A., Couroussé, D., Dumas, C., Maingault, L.: Deep learning side-channel analysis on large-scale traces - a case study on a polymorphic AES. Cryptology ePrint Archive, Paper 2020/881 (2020). https://eprint.iacr.org/2020/881
35. Migliore, V., Gérard, B., Tibouchi, M., Fouque, P-A.: Masking dilithium. In: Deng, R.H., Gauthier-Umaña, V., Ochoa, M., Yung, M. (eds.) Applied Cryptography and Network Security, pp. 344–362. Springer International Publishing, Cham (2019)
36. Moody, D.: Status report on the third round of the NIST post-quantum cryptography standardization process. Nistir 8309, pp. 1–27 (2022). https://nvlpubs.nist.gov/nistpubs/ir/2022/NIST.IR.8413.pdf
37. Mujdei, C., Beckers, A., Mera, J.M.B., Karmakar, A., Wouters, L., Verbauwhede, I.: Side-channel analysis of lattice-based post-quantum cryptography: exploiting polynomial multiplication. Cryptology ePrint Archive, Paper 2022/474 (2022). https://eprint.iacr.org/2022/474
38. NewAE Technology Inc. [n.d.]. ChipWhisperer. https://newae.com/tools/chipwhisperer. Accessed 2022
39. Ngo, K., Dubrova, E.: Side-channel analysis of the random number generator in STM32 MCUs. In: Proceedings of the Great Lakes Symposium on VLSI (GLSVLSI '22) (2022). https://doi.org/10.1145/3526241.3530324
40. Ngo, K., Dubrova, E., Guo, Q., Johansson, T.: A side-channel attack on a masked IND-CCA secure saber KEM implementation. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2021**(4), 676–707 (2021). https://doi.org/10.46586/tches.v2021.i4.676-707
41. Ngo, K., Dubrova, E., Johansson, T.: Breaking Masked and Shuffled CCA Secure Saber KEM by Power Analysis, pp. 51-61, Association for Computing Machinery, New York, NY, USA (2021). https://doi.org/10.1145/3474376.3487277
42. Oder, T., Schneider, T., Pöppelmann, T., Güneysu, T.: Practical CCA2-secure and masked ring-LWE implementation. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2018**(1), 142–174 (2018). https://doi.org/10.13154/tches.v2018.i1.142-174
43. Perin, G., Chmielewski, Ł, Picek, S.: Strength in numbers: improving generalization with ensembles in machine learning-based profiled side-channel analysis. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2020**(4), 337–364 (2020)
44. Primas, R., Pessl, P., Mangard, S.: Single-trace side-channel attacks on masked lattice-based encryption. In: Fischer, W., Homma, N. (eds.) Cryptographic Hardware and Embedded Systems - CHES 2017, pp. 513–533. Springer International Publishing, Cham (2017)
45. Ravi, P., Bhasin, S., Roy, S.S., Chattopadhyay, A.: On exploiting message leakage in (few) NIST PQC candidates for practical message recovery and key recovery attacks. Cryptology ePrint Archive, Report 2020/1559 (2020). https://eprint.iacr.org/2020/1559
46. Ravi, P., Deb, S., Baksi, A., Chattopadhyay, A., Bhasin, S., Mendelson, A.: on threat of hardware trojan to post-quantum lattice-based schemes: a key recovery attack on saber and beyond. In: Batina, L., Picek, S., Mondal, M. (eds.) Security, Privacy, and Applied Cryptography Engineering, pp. 81–103. Springer International Publishing, Cham (2022)
47. Ravi, P., Roy, S.S., Chattopadhyay, A., Bhasin, S.: Generic side-channel attacks on CCA-secure lattice-based PKE and KEMs. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2020**(3), 307–335 (2020). https://doi.org/10.13154/tches.v2020.i3.307-335

48. Reparaz, O., de Clercq, R., Roy, S.S., Vercauteren, F., Verbauwhede, I.: Additively homomorphic ring-LWE masking. In: Post-Quantum Cryptography, pp. 233–244. Springer (2016)
49. Reparaz, O., Roy, S.S., Vercauteren, F., Verbauwhede, I.: A masked ring-LWE implementation. In: International Workshop on Cryptographic Hardware and Embedded Systems, pp. 683–702. Springer (2015)
50. Schneider, T., Paglialonga, C., Oder, T., Güneysu, T.: Efficiently masking binomial sampling at arbitrary orders for lattice-based crypto. In: Public-Key Cryptography – PKC 2019, pp. 534–564. Springer International Publishing (2019)
51. Shor, Peter W.: Polynomial-time algorithms for prime factorization and discrete logarithms on a quantum computer. SIAM Rev. **41**(2), 303–332 (1999)
52. Sim, B-Y, Kwon, J., Lee, J., Kim, I-J., Lee, T., Han, J., Yoon, H., Cho, J., Han, D-G.: Single-trace attacks on the message encoding of lattice-based KEMs. Cryptology ePrint Archive, Report 2020/992 (2020). https://eprint.iacr.org/2020/992
53. Timon, B.: Non-profiled deep learning-based side-channel attacks. Cryptology ePrint Archive, Paper 2018/196 (2018). https://eprint.iacr.org/2018/196
54. Ueno, R., Xagawa, K., Tanaka, Y., Ito, A., Takahashi, J., Homma, N.: Curse of re-encryption: a generic power/EM analysis on post-quantum KEMs. IACR Trans. Cryptogr. Hardw. Embed. Syst. **2022**(1), 296–322 (2021). https://doi.org/10.46586/tches.v2022.i1.296-322
55. Ueno, R., Xagawa, K., Tanaka, Y., Ito, A., Takahashi, J., Homma, N.: Curse of re-encryption: a generic power/EM analysis on post-quantum KEMs. Cryptology ePrint Archive, Report 2021/849 (2021)
56. Veyrat-Charvillon, N., Medwed, M., Kerckhof, S., Standaert, F-X.: Shuffling against side-channel attacks: a comprehensive study with cautionary note. In: Wang, X., Sako, K. (eds.) Advances in Cryptology – ASIACRYPT 2012, pp. 740–757. Springer, Berlin, Heidelberg (2012)
57. Wang, H., Dubrova, E.: Tandem deep learning side-channel attack against FPGA implementation of AES. In: Proceedings of IEEE International Symposium on Smart Electronic Systems (iSES), pp. 147–150 (2020)
58. Wang, J., Cao, W., Chen, H., Li, H.: Practical side-channel attack on masked message encoding in latticed-based KEM. Cryptology ePrint Archive, Paper 2022/859 (2022). https://eprint.iacr.org/2022/859
59. Wang, R., Ngo, K., Dubrova, E.: A message recovery attack on LWE / LWR-based PKE / KEMs using amplitude-modulated EM emanations (2022)
60. Wang, R., Ngo, K., Dubrova, E.: Side-channel analysis of Saber KEM using amplitude-modulated EM emanations. In: 2022 25th Euromicro Conference on Digital System Design (DSD), pp. 488–495. IEEE (2022). https://doi.org/10.1109/DSD57027.2022.00071
61. Welch, B.L.: The generalization of 'student's' problem when several different population variances are involved. Biometrika **34**(1-2), 28–35 (1947). http://www.jstor.org/stable/2332510
62. Xu, Z., Pemberton, O., Roy, S.S., Oswald, D., Yao, W., Zheng, Z.: Magnifying side-channel leakage of lattice-based cryptosystems with chosen ciphertexts: the case study of kyber. Tech. Rep. (2020). https://doi.org/10.1109/TC.2021.3122997
63. Yu, Y., Moraitis, M., Dubrova, E.: Why deep learning makes it difficult to keep secrets in FPGAs. In: Proceedings of Workshop on DYnamic and Novel Advances in Machine Learning and Intelligent Cyber Security (DYNAMICS '20) (2020). https://doi.org/10.1145/3477997.3478001